

Mobile Game Development with Unity

Case: Project Runner

Kannisto, Kimi

2015 Kerava

Laurea University of Applied Sciences
Kerava

Mobile Game Development with Unity Case: Project Runner

Kimi Petteri Kannisto
Business Information Technology
Bachelor's Thesis
February, 2016

Kannisto, Kimi

**Mobile Game Development with Unity
Case: Project Runner**

Year	2016	Pages	50
------	------	-------	----

The purpose of this bachelor's thesis is to function as an introduction to the tools used in the development of the game 'Project Runner', and to give a case example of using those tools. The thesis also represents the author's skill in video game development. Project Runner was to be a side-scrolling runner game with a four person multiplayer and the ability to modify the terrain of the course by drawing shapes on the screen. This thesis details the development of the game and programming of the mechanics, while also teaching the basics of the Unity 3D game engine, as well as the Photon Unity Networking package that was used to implement the multiplayer.

The first half of the thesis introduces the tools and goes through the basics of utilizing them, while its second half goes through the phases of development and gives detailed descriptions of the written code that was used to achieve the mechanics of the game. The final result of the project was a working alpha-build of the game, with all of the planned core mechanics implemented. When the game was play-tested it was concluded that the game idea itself wasn't entertaining, and the project was cancelled. While the game idea itself may have proven to not be as interesting as originally thought, all of the game mechanics were successfully coded, so the project still functions as a good case example of achieving the specific mechanics themselves.

Keywords: mobile games, game programming, unity, photon, multiplayer

Kannisto, Kimi

Mobiilipelin kehittäminen Unityllä
Case: Project Runner

Vuosi	2016	Sivumäärä	50
-------	------	-----------	----

Opinnäytetyön tavoitteena on toimia perehdytyksenä Project Runner pelinkehityksen aikana käytettyihin työvälineisiin ja antaa käytännön esimerkki niiden käytöstä. Project Runner on sivultapäin kuvattu juoksupeli, jossa kilpaillaan kolmea toista pelaajaa vastaan. Pelin erikoisuutena on kyky muokata juoksuradan maastoa piirtämällä ruutuun tiettyjä muotoja.

Tässä opinnäytetyössä käydään läpi pelinkehitysprosessi, pelimekaniikkojen ohjelmointi ja projektissa hyödynnetyt työkalut Unity 3D ja Photon Unity Networking. Unity 3D on pelimootori, jolla peli kehitettiin. Photon Unity Networking on koodipaketti Unitylle, jonka avulla voi luoda nettipelitoimintoja. Työkaluista käydään läpi niiden taustatieto, perusteet, sovellus ja niiden antamat hyödyt ohjelmoijalle. Opinnäytetyö antaa myös tarkat kuvaukset kirjoitetuista koodista, joilla pelimekaniikat saavutettiin.

Projektin lopputuloksena oli toimiva alpha-versio pelistä, johon sisältyi kaikki suunnitellut päämekaniikat. Peliä testattaessa todettiin, että itse peli-idea ei ollut mielenkiintoinen ja projekti lopetettiin. Vaikka peli-idean itsessään todettiin olevan odotettua tylsempi, kaikki pelimekaniikat olivat onnistuneesti ohjelmoitu, joten projekti toimii edelleen hyvänä tausesimerkkinä kyseisten mekaniikkojen saavuttamisesta.

Asiasanat: mobiilipelit, peliohjelmointi, unity, photon, moninpeli

Table of Contents

1	Introduction	8
1.1	The Development Process	8
1.2	The Project	9
1.3	Game Outline	10
2	Utilized Tools	12
2.1	C# Programming Language	12
2.2	JavaScript Programming Language	13
2.2.1	JavaScript's Differences in Unity	13
2.3	Unity 3D Game Engine	14
2.3.1	Basics of Unity	15
2.3.2	Scripts in Unity	19
2.3.3	Accessing Other Scripts and Game Objects	20
2.4	Photon Unity Networking Package	22
2.4.1	PUN Components	22
2.4.2	Setting Up the Server and Game-Room	23
2.4.3	Creating a Room and Photon's Preset Functions	24
2.4.4	Remote-Procedure-Calls	25
2.4.5	Coding a Data Synchronizer	26
3	Developing the Game	27
3.1	Unity Built-in Functions and Other Terminology	27
3.2	The Graphics and Audio	29
3.3	Development Phases	29
3.3.1	Detecting the Drawn Shape	30
3.3.2	Adding the Terrain	30
3.3.3	Dashing Through Objects	31
3.3.4	Looping the World	32
3.3.5	More Advanced Draw Detection and Controls	32
3.3.6	Multiplayer	33
3.3.7	Alternate Version and Ending the Project	34
3.4	Game Overview	35
4	The Game Code	37
4.1	Game Specific Mechanics and Terms	37
4.1.1	Symbols, Traps and Terrain	38
4.1.2	Special Traps and Actions	38
4.2	Setting Up the Project	38
4.2.1	Scenes	39
4.2.2	Prefabs	39

4.3	Scripts	40
4.3.1	GameController.cs and MenuArranger.cs	40
4.3.2	Matchmaker.cs and NetworkPlayer.cs.....	41
4.3.3	PlayerScript.cs and PlayerClone.cs	42
4.3.4	TouchInput.cs for Detecting Screen Touches	43
4.3.5	ActionScript.cs for Handling Buttons	44
4.3.6	MoveAtBoundary.cs and Object Duplication in GameController.cs	45
4.3.7	SymbolButton.cs and Trap Addition in GameController.cs	47
5	Conclusions and Author's Comments on the Project	49
	References	51

TERMS

Runner	A game where the player controls a character that runs through some type of an obstacle course
Side-scroller	A type of 2D game where the player character is depicted from the side, with the game world scrolling left or right when moving
Script	A file containing a series of code that is run by the program
Game Engine	A software framework that is used to develop video games
Alpha-build	The alpha phase of the release life cycle is the first phase to begin software testing
Backend	The backend is the "data-access" layer of the software, while the frontend is the user-friendly interface that accesses the backend
Serialization	Serializing is the conversion of an abstract data-type to a series of bytes
Cast	A cast is an operator that will convert a variable's data-type
Switch case	A type of loop that checks for matches. A variable is given as a parameter and it checks if the value is equal to a "case"
Coroutine	A function that is run simultaneously with the rest of the running code in a separate thread

1 Introduction

'Project Runner' was a mobile game being developed by Digital Hammer. The members of the development team were Marcus Dake as the graphics designer and the author of this bachelor's as the programmer. While working as the programmer for the game, the author had to learn how to use the Unity 3D game engine as well as the C# programming language. The concept for the game was a side-scrolling runner game with a four person multiplayer, where the unique aspect was the ability to change the terrain by drawing shapes on the screen. The game was to also have a distinct aesthetic style based on the art of a game the author had made for an earlier, smaller game. All of the author's previous programming experience was with Java, and only included minor projects done throughout university.

The purpose of this thesis is to teach the basics of developing a mobile game with the Unity engine, specifically for the Android platform, as well as give a practical example in the form of Project Runner. The thesis will go through the Unity engine itself, the Photon Unity Networking package which was used for the creation of the multiplayer, and the development of the project with detailed descriptions of the scripts that were used to achieve the various mechanics. The hope is that the thesis could work as a form of tutorial to get a person started with Unity and Photon, as well as a representation of the author's ability as a programmer.

The thesis will not cover general game development theory, as that is a subject that could fill an entire thesis all on its own, and is not necessary for the goals of this thesis. Elements of Unity and PUN that are introduced are also limited to just the basics that are needed to get started, and ones that were relevant to the development of Project Runner. Sound and art design are also excluded from the topic of this thesis.

1.1 The Development Process

While a methodology for the development of Project Runner wasn't specifically defined, the project took the agile development approach of incremental iteration and continuous feedback, keeping the development process flexible. Certain requirements for the next version would be defined, as well as some type of a deadline for when the version should be in a presentable condition. The version was then looked over and the next step was decided. As such, the methodology was similar to scrum, but lacking any type of 'scrum master' and requirements could change in the middle of a version's development.

The entire development process from beginning to end took an approximate two months, though this still includes some down time, typically caused by waiting on feedback for a spe-

cific version. Most of the core mechanics for the singleplayer were programmed over the course of three to four weeks, while the initial implementation of the multiplayer and network mechanics was done in one. The last two weeks involved refinement, optimization, bug fixing and play-testing, as well as the addition of a few more minor mechanics. The version handling for the project was done with the program 'Source Tree', which can be used to commit versions to GitHub. Since all of the work was done remotely from home, excluding some meetings that were held at the company, communication between project members was largely accomplished through the website 'Slack', while Google Drive was used for the sharing of files like artwork and documents. The game versions would usually be shared by uploading them to Dropbox and giving the public download link through Slack.

1.2 The Project

During a company meeting between the new interns and the CEO, several game ideas for a mobile platform were brainstormed. Amongst the ideas presented was a relatively standard 'Runner' type game, but with a few selling points. The game's art style was to be based on the dark and sketchy look of one the author's earlier games, JavaQuest II (Figure 1), with 'shadowy' stick figures as the runners. The style was considered unique, and also fairly simple and quick to accomplish. The game itself was to be a multiplayer game where the player races against other human opponents, with a variety of unlockable character's and abilities. The 'unlockables' would form the financial model of the game. While the game itself was to be free, and all unlockable characters and abilities could be gained by simply playing the game enough, the player could also use money to buy them to get them faster. The final unique point of the game was its core game mechanic. The player could draw symbols on the screen to modify the game world, to either create slopes for himself or to set traps for the other players. The world would loop eventually, so any obstacles the players created would also appear in front of them after some time.



Figure 1: Two 'JavaQuest II' art style examples

The idea was found interesting, and most importantly, unique. Nobody knew of any other game with such mechanics, so it was considered to at least be worth a try. Since Marcus Dake and the author were the originators of the concept, they were tasked with its completion. The first step upon starting work on the project, was to create a more detailed concept. If the company was to try and make money with the game by having purchasable characters and abilities, it was necessary to be able to come up with enough of them. It was also necessary to go into more detail about the actual mechanics of the game and how to balance them.

1.3 Game Outline

The player controls one character. The world scrolls automatically to the left at a set pace, preventing the character from ever stopping. Should the character run into an obstacle that is too steep for it to climb, such as a wall, the character would stumble and slow down, but go through the obstacle. The character also has the standard ability to dash forward, which has a 2 second cooldown before being possible to use again. During the dash, the character can pass through anything that is marked as an obstacle. All the characters are also able to jump and create terrain. For the terrain creation, there is an area in the middle of the screen that detects the shape you draw, and in accordance to that shape, chooses which type of terrain is to be added to the world. The terrain options included mountains, slopes, floating platforms and spikes that rise from the ground to function as obstacles.

At the start the team designed four unique looking characters, and made up a variety of ‘special abilities’ that the player could pick. The character appearance was to be purely aesthetic. The abilities included a ‘double-jump’ that would allow the character to jump once more while in the air for greater height, a ‘tackle’ that would cause the dashes to destroy obstacles and cause other players to stumble, and a ‘grappling-hook’ that would shoot up and forwards, latching unto any surface and pulling the character to that point. The character designs included the lean ‘runner’ (Figure 2), a female ‘ninja’, a bulkier ‘tank’ and a ‘witch’ that hovers forward instead of running.



Figure 2: Concept art of the standard ‘Runner’ character by Marcus Dake

Later on the team also added ‘special’ terrain objects that the player could create by simply pressing the corresponding button, but had a cooldown. An example of such was the ‘cage’ that would form a circular cage for 2 seconds around whoever was currently in first place. Another would fire a spike forwards, causing anyone hit by it to stumble, and one was the ‘warp’, which created a wall of light that, when passing through it, would propel the character forwards in a super-dash. These were only to be available to players who weren’t in first place, to allow them to better be able to catch up to whoever was, and keep the race more interesting.

There was also the idea for visual objects that the players could add to their characters for a more unique look (scarves, hats et cetera), but this was merely something in the planning

that never got implemented. There were also further ideas for characters and abilities, but these were the ones that we decided to try to get into the game first to test out the mechanics for choosing them. Once the game itself was working, more variety could be added easily enough later on.

2 Utilized Tools

Several tools were used throughout the project, most central of which were the Unity 3D game engine and the Photon Unity Networking package. Unity 3D is the tool with which the core game was programmed, and the Photon Unity Networking package is a downloadable collection of code for Unity that allows the implementation of network functions to the game. The chapter also includes a few short words about the C# and JavaScript programming languages, either of which can be used for programming in Unity.

2.1 C# Programming Language

C# (pronounced 'see sharp') is a programming language developed by Microsoft and released to the public back in June of 2000, making it a relatively recent programming language. Microsoft developed it as a platform-independent language in the style of Java, intended as an improvement on the C and C++ programming languages, trying to eliminate many of their problems while keeping their strengths. The result is a strongly-typed, object-oriented programming language that people with prior experience in Java will find oddly familiar. This sense of familiarity stems from the fact that C# is in fact very similar to Java, with various aspects of it being identical, making it very easy to pick up if you've ever done anything with Java before. (Allain 2011, Jones 2001.)

C# attempts to remove many of the complexities and other issues that existed in Java and C++, such as macros, templates, multiple inheritance and virtual base classes. These were all problematic areas that tended to cause confusion, but are entirely removed and replaced in C#. Various redundancies that also existed in the older C languages have also been removed, making the language simpler while still keeping operators and statements largely the same as before. C# also added a lot of modern functionality to the language such as exception handling, garbage collection and code security. (Jones 2001.)

C# is designed to be modular, with code written in chunks referred to as 'classes', with functions inside the classes (called 'member methods') being easily reusable by other programs. It also supports 'encapsulation' (situating functionality into 'packages') and 'inheritance' (expanding existing code into other packages) due to its object-oriented nature. As a strongly-typed programming language, all variables have to have a 'data-type' defined, and it will not

allow usage of, for example, a decimal number in a variable defined as 'int' (integer). While this can be limiting, it simplifies certain functions such as operators and reduces the risk of mismatching different data-types in situations where they shouldn't be. (Jones 2001.)

2.2 JavaScript Programming Language

In 1995 Netscape Communications Corp wanted something to help in handling the input validation that had up until then been left to server-sided languages such as Perl. At the time, it had been necessary to go to the server itself to check whether a necessary field had been left unfilled. To make the process simpler, JavaScript was invented, and it has since then developed into a vital part of all major web browsers, interacting with almost every aspect of the browser's window and content. (Media.wiley.com 2014.)

JavaScript is a cross-platform, object-oriented scripting language. Its primary usage is in the creation of interactive web pages, particularly things like quizzes and polls, and it is run on the client's side as opposed to the server, thusly not requiring constant downloads from the server. One of its major differences to C# is the lack of distinction between types of objects, and essentially any properties and methods can be attached to any object. Another interesting aspect of the language is the possibility for functions to be object properties. JavaScript is a free-form language, where you do not have to declare all of the variables, classes and methods, nor be too concerned with whether things are public, private or protected. (Chapman 2015, Mozilla 2015.)

2.2.1 JavaScript's Differences in Unity

Although JavaScript itself uses dynamic-typing, not requiring a definition for the data-type of the variable or object, using JavaScript in Unity is a little different. When declaring a variable without setting a value to it, it actually is necessary to define a type for it. This is done by adding ' : theTypeOfTheVariable' to the end. For example, 'var text : string'. However, this is not necessary when setting the variable's value in the same statement that it is being declared in. Unity will automatically infer the type from the given value. So if instead there is the line of code 'var text = 'Some text.'', the type of the variable will automatically be defined as a 'string' of text. The type can still be declared manually for clarity however. (tonyd 2009.)

Another thing to note about variable types, is that it is not possible to change the value of the variable to something that isn't of the same type. This will require conversion commands (such as 'toString' to translate the value into a string). Once again going with the earlier example, after having defined 'text', the value cannot be changed to 5, since 5 is an integer

value and not a string value. Due to this, Unity does remove some of the freedom involved with regular JavaScript coding. (tonyd 2009.)

2.3 Unity 3D Game Engine

The Unity 3D game engine is the world's leading software for the development of video games, with support for 22 different platforms from Linux to Xbox, and 4,5 million registered users worldwide. Unity Technologies, a company founded in Denmark in 2004 and the developer of the Unity engine, boasts a 45% market share, with over 600 million people across the world playing video games created with the aid of the Unity engine. Unity is used in the creation of 50% to 75% of the games on the mobile gaming market, depending on the region. The customers of Unity Technologies include companies such as Coca Cola, Cartoon Network, Microsoft, NASA and Ubisoft, and included in the list of games developed with Unity are games such as Blizzard's 'Hearthstone' and Colossal Order's 'Cities Skylines'. (Unity Technologies 2015.)

The driving principle behind Unity Technologies' development of the game engine was the goal of making video game development as accessible to as wide of an audience as possible, with the hopes of promoting individual creativity by providing an easy-to-use program. Unity hosts a vast variety of tools and built-in functions to simplify various aspects of game development and to reduce tedium. For example, if the user wishes to have their game made into both an iOS and Android version, they simply change the project's target platform and Unity will convert the game for them. The user can build the entirety of the game and its file structure inside the engine and it will build the runnable files. These functions, alongside various instances of code that the engine can provide premade (such as code for a 'joystick' type controller) are very valuable to someone wishing to jump into game programming, without having to work out the basic things like graphics drawing or physics. Physics in particular can be complicated and tedious to do from the ground up with something like Java and a simple text editor. All the tools needed for adding graphical elements, buttons, animations and physics are included in the engine, so the user can skip right away to programming the actual game mechanics. (Unity Technologies 2015.)

In addition to the tools provided in the program itself, Unity Technologies also provides the 'Asset Store'. The asset store is an online webstore in the vein of Google Play and the like, where users of Unity can upload their self-made code packages to provide even more tools for other users. A prime example of such a package is the Photon Unity Networking package. Someone was not satisfied with the built-in network functionality of Unity, so they wrote their own networking code, packaged it and placed it into the asset store for the use of oth-

ers. Packages are extracted into the game's file directory, adding the code to the specific project.

As for the writing of code itself, Unity supports two programming languages in addition to its own Unityscript. C# and JavaScript. Both very commonly used and both quite easy and straightforward. The user can create their files with either one, but there are some things that need to be kept in mind when it comes to using both JavaScript files and C# files in the project and wanting them to have access to each other. When the game is run, Unity compiles all of the scripts, and C# script files are compiled first. Because of this, JavaScript files can access the contents of C# files just like other JavaScript files, but the reverse is not true. In order to have a C# file access the code of a JavaScript file, the user has to modify the load order. The way to do this, is to have the JavaScript file the user wants to access via C# in a folder named 'Plugins'. 'Plugins' is one of the 'special folders' that Unity checks for during compilation. Any file in 'plugins' will be compiled before any of the other files. Thus, a JavaScript file in 'plugins' will have been compiled already by the time C# start compiling, and can be accessed freely. (Unity Documentation.)

2.3.1 Basics of Unity

Everything in Unity is a game object with components attached to it. A game object itself is an empty container, and by adding components to it the user is adding to its functionality. When writing a script in either C# or JavaScript, the script alone will never do anything in the game. What the user is actually creating when writing the code for a script is a component that can be attached to any game object within the game world. The goal of this setup is to provide maximum modularity for the code. So the user could write the file 'healthScript.cs' and code into it a publicly accessible value for the 'hit points', and add a public function that reduces those hit points. They can then add this script as a component to any game object that they want, and that game object will contain its own instance of that script, with each separate instance having its own hit point value. Unity also provides a vast array of its own components, the most common of which is the Transform component. This component comes with every new game object created and cannot be removed. The Transform component defines the object's location in the game world, and can be used to move, resize or rotate it. All game objects created into the current game world can be found in the game object hierarchy window (Figure 3).

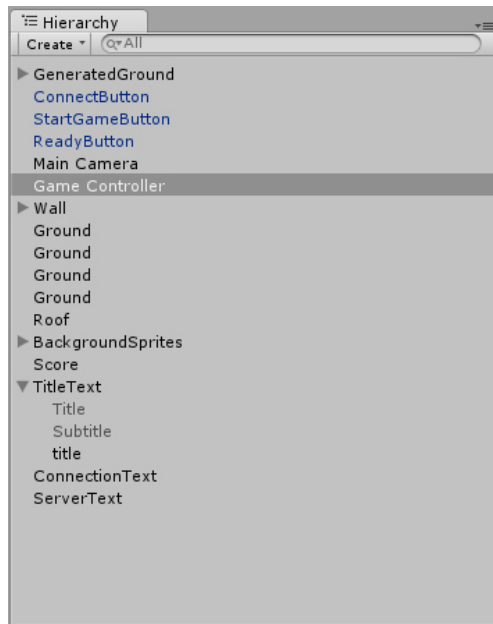


Figure 3: The game object hierarchy showing all of the game objects in the game world

To allow easy utilization of these components, Unity provides the 'Inspector' window, by default found on the right side of the program. When selecting a game object from the hierarchy of the game objects currently in the game world, the inspector will show all of the game object's components, with edit fields for every variable in the component's scripts that have been defined as 'public' variables (Figure 4). This allows the user to easily edit the value for each individual game object. Going back to the 'healthScript.cs' example, if the script defines the 'hit points' (HP from here on) as 100, the inspector will show the HP variable with an edit field next to it reading 100. This value can then be modified in the inspector so that that specific game object has a different amount of HP.

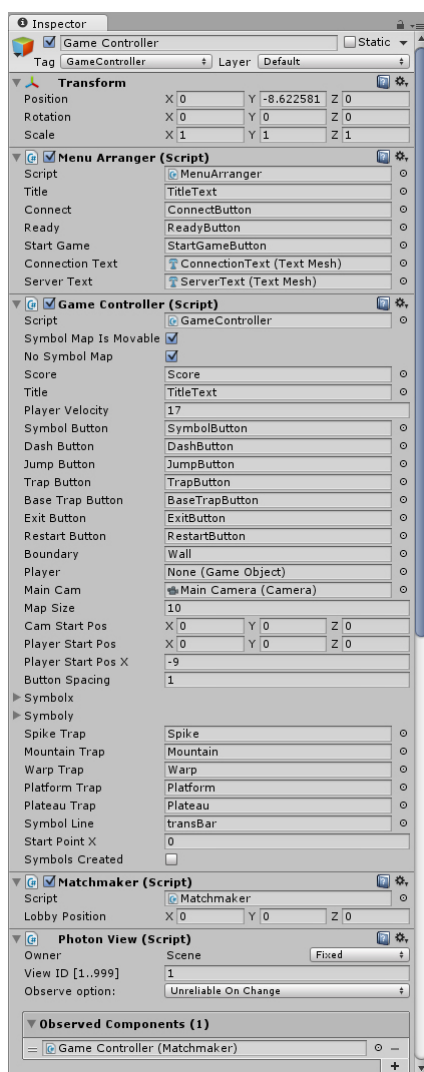


Figure 4: The Inspector showing the components attached to the Game Controller game object

Created game objects can also be 'saved' so that it is not necessary to always manually create a new one, attach the components and assign the custom values whenever another identical one is needed in the game world. This can be accomplished by simply dragging any game object from the hierarchy to the project directory (Figure 5). This will save the game object as what is called a 'prefab'. Any saved prefab can then be dragged from the directory to either the hierarchy or directly to the game world window, and Unity will create a 'clone' of that prefab. If a script has a 'GameObject' type variable in it that is set to public, a prefab can also be dragged in to the variable's corresponding field in the inspector, and that prefab will be assigned to that variable. This way the script can be used to create clones of a predefined game object. Any game object that is made from a prefab will also have an extra bar in the inspector that allows the game object to be reverted back to its default values as defined in the original prefab, or to have any changes that have been made to the clone be applied to the prefab it is a clone of.

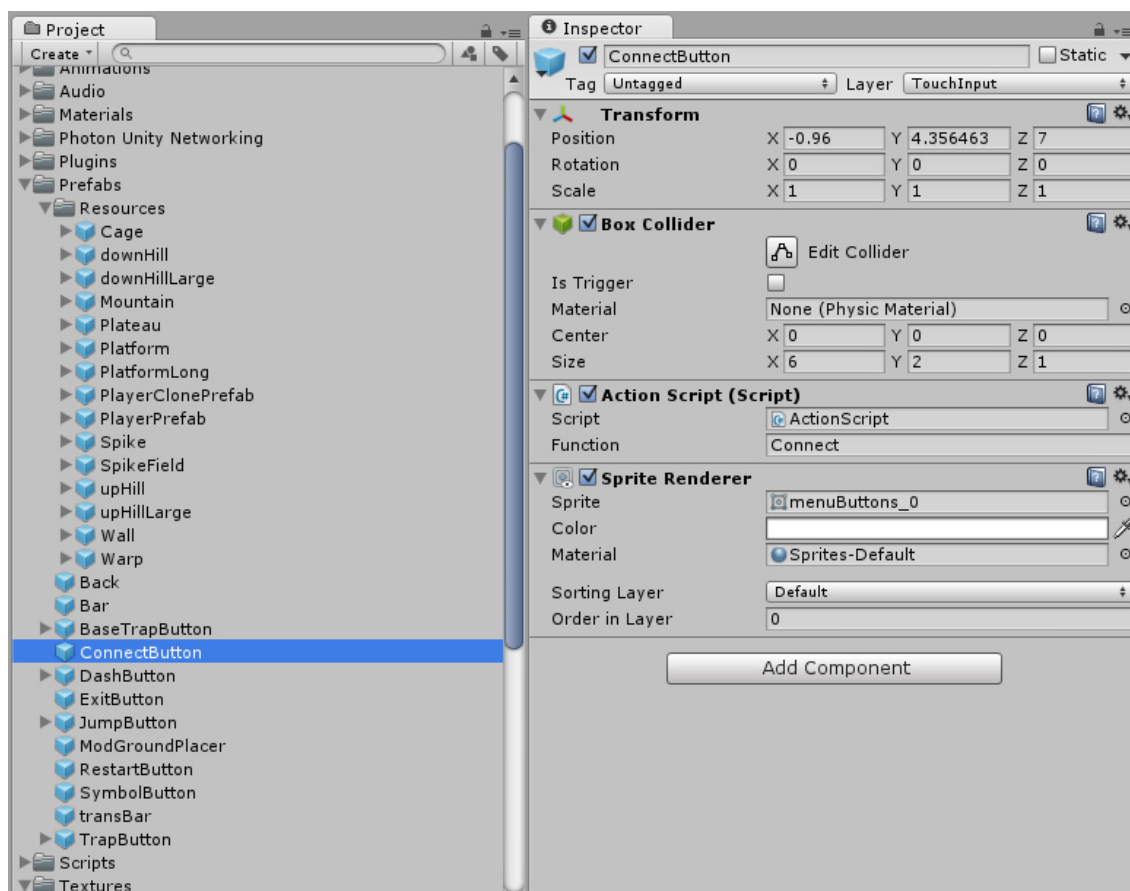


Figure 5: The folder hierarchy with saved prefabs. Prefabs can also be edited in the inspector.

Unity has a lot of components readily available for use, as well as options for adding game objects that come with specific components already attached. These objects include lights that come with components relevant to creating light sources in the game world, 3D objects that come with components that draw a specific 3D shape at the object's location and a 'collider' that allows the shape to interact physically with other objects that also include a collider. The game will also create a 'camera' object whenever a new 'scene' is created that functions as your 'game view'. When the game is actually played, what is 'seen' by the camera is what is shown to the player on the game screen.

Another important aspect are 'scenes'. Each scene in unity is a composition of game objects. When creating a new project, Unity will create a new scene for the project, and any game objects created are added to that particular scene. When creating a new scene, the new scene will have none of the game objects that were in the other, and no additions to it will add anything to the other scenes. Scenes can be saved in the project's folders and can be loaded through the script. An example of using this would be having a different scene for the game's start menu, and the actual game world itself. The game menu scene holds all the game objects and components required for changing game options, exiting the game and starting a

new game. When clicking on the 'start game' button, a script attached to that button can be made to load the 'main' scene, where all the gameplay itself is. This can also be used to divide the game into 'levels', so that the entire game world doesn't have to be loaded all at once, reducing computer memory consumption.

What actually happens when loading a scene, is that Unity will 'destroy' all of the game objects currently loaded, and load all the game objects of the new scene. If the user wants something to carry over from one scene to the other (for example game settings like a difficulty setting), game objects can be defined to not be destroyed when loading a new scene with `'DontDestroyOnLoad(someGameObject);'`. This will prevent the defined game object and any of its components or child objects from being destroyed when loading a new scene. An alternative way of having variables and such carry over, is using a static variable. If a script is defined as a static class, its usage will be more akin to the code files in standard programming such as Java. A static script cannot be added as a component to any game object, as static classes cannot have multiples, but a static class will exist outside of the scene, being always freely accessible. Any changes made to static variables will persist so long as the game is running, and the variables can be called by any component in any scene, allowing any of them to modify the value of the variable. (Unity Documentation.)

By default, all game objects created are independent of each other, and moving one of them will not move the others. However, the Transform component also includes the function of 'parenting' other Transform components. What this means is that a Transform set as a 'child' of another (`Transform.parent = otherGameObject.Transform`) will no longer display its absolute position in the game world, and instead show its position relative to the parent. Moving the child will not move the parent, but moving the parent will move the child as well. This is called a 'Transform hierarchy', and is very useful for tying a game object's position to another. For example, let's say there was a button, and a text mesh telling what the button does. The text is intended to always be a few pixels to the right of the button, and to move alongside the button. This can be achieved by creating an empty game object, and making both the button and the text a child of the empty game object. This way, in order to move both simultaneously, it is only necessary to move the empty parent. It is also possible to simply make the text directly a child of the button, or vice versa. Child objects can also be set as parents for other objects, allowing for a multilevel hierarchy of children (Unity Documentation.)

2.3.2 Scripts in Unity

Unity has various functions that become available for use in a script when the line `'using UnityEngine;'` is placed at the top of the file (which is done automatically for all new scripts).

The default ones that come with new script files are the `'Start()'` and `'Update()'` functions. Another one to also be aware of is the `'Awake()'` function. If the script has a function that is simply defined as `'void'`, with one the aforementioned names following it, Unity will handle the automatic calling of those scripts. `'Awake()'` is called the instant the script component is instantiated into the game world. Because `'Awake()'` is possible to run before the other components of the game object being created have finished instantiation, `'Awake()'` is not in the files by default, and instead the default is `'Start()'`. `Start()` is called when the entire game object has finished instantiation, and is typically used to define the variables of the script and assign other default values. If there is some code that the component is intended to run the moment it is created, this is where that code is placed. `Update()` on the other hand is called at every frame of the game. Due to this, one shouldn't put any taxing code into the `Update()` function, as this code will get run a lot of times per second, particularly if there are multiple game objects with the same taxing code in their own `Update()` functions. There is also the `FixedUpdate()` function, which is recommended for physics related code. `Update()` is called every frame, and as such, it will get called more or less often depending on the framerate the game is running on, while `FixedUpdate()` is run at the same intervals regardless of the game's framerate. (Unity Documentation.)

There are a lot more functions that Unity calls on its own when it finds them, and many of them are dependent on other components being attached to the game object in order to ever get called by the Unity engine, such as the `'OnCollisionEnter(Collision collision)'` function, which is called when the current game object's collider detects a collision, and will receive as a parameter the Collision type variable `'collision'` that contains information about the collision and the game object colliding with the collider. The best way to find out about more of such functions is to refer to the Unity documentation. (Unity Documentation.)

2.3.3 Accessing Other Scripts and Game Objects

For someone coming from standard software programming with something like Java, Unity's object oriented programming can feel a little unusual at first, and it can take some time getting used to thinking in terms of objects. In Java for example, the programmer writes files into folders and all of the files are easily accessible to the others. Unity's way of doing things is quite different from that, but once the programmer starts getting used to it, creating modular code is very easy. However, because of the modular nature of Unity, certain simple things will at first seem tricky to do. One obvious one is `'how to call a function from another script'`. In Java this is quite easy, so long as the file that has the function one wants to call is in the package their script incorporates. In Unity however, something as fundamental as this, while not actually complicated, does require a bit more work.

Since scripts are components attached to game objects, they are generally speaking ‘unaware’ of each other. Calling another script that is attached to the same game object is still fairly easy to do however. Let’s say there was a script that wanted to call the ‘reduceHealth()’ function in ‘healthScript.cs’, and the health script is attached to the same game object as the script that is supposed to use the function. The script should have a new variable added with the data-type ‘healthScript’, which in this case will be named ‘hs’. The variable can then be made into a reference to the healthScript.cs component with ‘healthScript hs = GetComponent<healthScript>();’. The ‘GetComponent’ function will get any script attached to the Transform of the script’s game object that has the same name as the defined parameter. Once this is done, the health reduction function can be called with ‘hs.reduceHealth();’ and it will run the function in the health script. Alternatively, in case the programmer does not want to define a variable as a reference to the script, it is also possible to just write ‘GetComponent<healthScript>().reduceHealth()’. While this will work just as well, it should be kept in mind that ‘GetComponent’ is a search function that loops through all of the components. If the script is going to be doing repetitive calls to functions and variables in the health script, it will be less taxing for the program to have to only do the search once when assigning the variable, than having to do it multiple times. But, if the script is going to only call it once, not assigning a variable to it will on the other hand free up memory usage. (Unity Documentation.)

If the health script that is supposed to be accessed is attached to an entirely different game object however, GetComponent by itself won’t do anything. At this point there are two primary options. One is, once again, a variable. If it’s known that a script is going to be accessing another game object a lot (for example a ‘game controller’ object that will contain a lot of the code for running the game itself), the best option is defining a public ‘GameObject’ type variable in the code. After this, the specific game object can be dragged from the hierarchy to the component in the Inspector. Once the component has a reference to the game object in a variable, that shall in this example be named ‘go’ (for Game Object), ‘go.GetComponent<healthScript>()’ can be used to get access to the health script attached to that specific game object. But what if the game object does not exist at the beginning of the game? What if it’s a clone made from a prefab during the game’s run time (for example a bullet fired from a gun)? (Unity Documentation.)

At this point, there is no ‘light’ way of accessing the game object, as the only option is to loop through every single game object in the game until the one that is wanted comes up. Unity luckily already has its own code for doing this, with a few varying versions. Unity’s class ‘GameObject’ contains two functions for finding game objects based on their ‘tag’. Tags are additional names given to game objects. Most game objects by default are simply tagged as ‘Untagged’, but certain things like the camera are by default tagged as the ‘MainCamera’.

Unity has a few preset tags, and new ones can be easily added as well. The functions available are `'GameObject.FindGameObjectWithTag(tag)'` and `'GameObject.FindGameObjectsWithTag(tag)'`. The first one will loop through all the game objects and return the first one with a tag that matches the string given as a parameter. The latter will return an array with all of the game objects with tags that match the parameter. So, one line that I've written quite a few times now that gives a script access to the `'GameController.cs'` script in the `'Game Controller'` game object goes as follows: `GameController gc = GameObject.FindGameObjectWithTag('GameController').GetComponent<GameController>();`. That line finds the first game object tagged `'GameController'`, and gets the component in that game object named `'GameController.cs'`, and assigns it to the `GameController` type variable `'gc'`. Now any function in the game controller can be called from this script with `'gc.anyPublicFunction()'`. (Unity Documentation.)

2.4 Photon Unity Networking Package

Photon Unity Networking Package (or Photon PUN) is a downloadable package for Unity developed by Exit Games that allows easy usage of their `'Photon Cloud'` for free, for setting up cross-platform multiplayer into a game. It essentially provides all of the backend for multiplayer, allowing the developer to focus on coding the actual game itself. When installing the package, the user will be asked to create an account on the Photon Cloud page. Doing so will give them access to the cloud server for use in multiplayer. The free plan that's received automatically does have a limit on the amount of possible traffic (20 concurrent users), so when developing a much larger game that is supposed to handle massive amounts of concurrent users, an upgrade is likely warranted, but the free version was more than enough for the development and testing of Project Runner. Once the account is made, the user simply needs to input their App ID that Photon provides into the Unity package's settings, and they're ready to start. The thesis will go through the steps for doing so later in the chapter. (Photon Unity Networking Intro 2015, Photon PUN 2015.)

2.4.1 PUN Components

PUN comes with a fair few components for objects, some of which are necessary in order to make the object appear on the network. The most central of these components is the `'Photon View'` component. The component attaches the object to the user's `'view'`, which contains all of the network objects, each defined by an ID, and handles the transmission of information through the network. The view is handy in it that it can be used to easily tell if an object belongs to the user, or another person on the network (`PhotonView.isMine` will check for this). When adding the Photon View, it needs to be told what components should be observed. Information changes in these components will be sent to other players. For example, if there's

an object that can move around, and one wants the local changes to the position to be transmitted to other players, the 'Transform' component of the object would be added to the observed components. This way, everyone else's local 'version' of the moving object will receive their Transform data from the network and adjust themselves accordingly.

Simply adding the Transform is not necessarily enough though. Due to data packets being sent along the network at a certain rate, very fast movements will cause the update of the position for other players to appear 'jittery'. Let's say the game is running at 60 frames per second, and the object is moving 5 units every frame. But data packets are sent only 20 times a second. This way each update to the position coming in from the network will set the object to the new location, 15 units away from the original location (since the object has moved five units three times between the updates due to the framerate being three times larger). To the other player, the movement will not be a smooth transition from point A to B, but instead a 'teleportation' from A to B. In order to counteract this, it is preferable to use another PUN provided component, the 'Photon Transform View'.

The Photon Transform is specifically for the purpose of handling changes in the Transform component's data in a smoother way, giving a lot of ready options for interpolation and extrapolation between the old position and the updated position, and should be the observed component instead of the regular Unity provided Transform. It is also perfectly possible to write one's own synchronizer with one's own settings for location interpolation. There are also other PUN provided alternatives for things like Rigidbody components as well, for synchronizing the physics velocities of the object. All such components, including any self-written scripts for manually sending information (such as variables) should all be added to the object's Photon View's 'observed components'.

Typically all classes in Unity will inherit the MonoBehaviour that actually contains all of Unity's statements and syntax so the code can be recognized. In order to have a script recognize Photon related code and commands however, the classes need to inherit Photon.MonoBehaviour instead. Without this inheritance, anything like a remote-procedure-call or calls to the PhotonView will merely give error messages, since they do not exist in the standard MonoBehaviour.

2.4.2 Setting Up the Server and Game-Room

Before the user can start on coding the actual cross-network communication to other players, it is necessary to actually establish a connection to those players. The first step is to give the Unity project the App ID of the Photon Cloud account. This can be done by either going to 'Photon Unity Networking/Resources/PhotonServerSettings' in the folder hierarchy after add-

ing the package files, or to 'Window/Photon Unity Networking/Locate Settings Asset' in the Unity top bar. The inspector will open the file and the file has a few settings. 'Hosting' is Photon Cloud, 'region' is whatever region the user is in, and the 'AppId' field is where to paste the ID. Once this is done, PUN will be able to connect to the Photon Cloud servers through the Photon account.

The next step is to tell the game to actually make the connection. For this, it is recommended to create an entirely new script purely for handling all of the 'matchmaking' related code, as Photon comes with a lot of preset functions that are automatically called by Photon (just like the `Start()` and `Update()` functions in standard Unity) that the programmer would likely want to write the code for. It is also simpler to have a single component that holds all of the remote-procedure-calls, as they are called through the object's Photon View. The first thing to check for is whether or not a connection is already established. This is useful in cases of reloading the same scene. This information is held in `'PhotonNetwork.connectionStateDetailed'`. More specifically, it holds a detailed description of the current connection state, which will match `'PeerState.Joined'` when connected. So `'if(PhotonNetwork.connectionStateDetailed != PeerState.Joined)'` will check if the game is not connected. `connectionStateDetailed` can also be used to print out the status of the progression when the connection is being made, by printing it out at each frame. To start the connection process, all that is needed is to call `'PhotonNetwork.ConnectUsingSettings('versionIdentifier')'`. The `'versionIdentifier'` in this case is a string that is used to tell the version. If set to `'0.2'`, the game will not be able to connect with a player whose code has the setting at `'0.1'` or `'0.3'`. This way it can prevent older versions of the game from being able to interplay with newer ones, which could cause severe conflicts. Another setting that's set at this point is `'PhotonNetwork.automaticallySyncScene'`, which will sync things like the loading of scenes between players. After this is done, the game is now connected to the cloud, and the next step is to connect to other players on the cloud.

2.4.3 Creating a Room and Photon's Preset Functions

Photon multiplayer happens in what is called a 'room'. A room is a virtual construct that enables network communication between anyone in the same game session. While the game is now connected to the Photon Cloud, it is not connected with any other players yet. Photon automatically connects to a lobby during the connection process that holds all the rooms, and Photon has a preset function that is run when this happens named `'void OnJoinedLobby()'`. To have the game connect to a room after the connection to the lobby is established, the line `'PhotonNetwork.JoinRandomRoom()'` should be placed into this function. It will attempt to join a random room that exists in the lobby. However, if no one has yet created a room, the lobby will be empty, the connection will fail, and Photon will call the function `'OnPhoton-`

RandomJoinFailed()'. In this function, the game can order Photon to create an empty room with 'PhotonNetwork.CreateRoom(null)'. This will also automatically connect to the room, and the game can set the settings of the room. An example of a setting is 'PhotonNetwork.room.maxPlayers', which is an integer variable that defines the maximum connections to this room.

With the room created, the game is now being synchronized through the cloud with any other person who has connected to the same room. There are still a quite a lot of preset functions for various network events that can be useful to add to the code, but only ones used in Project Runner will be explained. OnMasterClientSwitched() is called when the master client (by default the creator of the room or the first person to connect) disconnects. Photon can be told to either select a new master client, or to simply restart the game. OnPhotonPlayerConnected() is called when a player connects to the room, and OnPhotonPlayerDisconnected() is called when one leaves. Both functions receive a PhotonPlayer type object as a parameter, which holds the information of the connected player, such as the network ID number. These functions are useful for keeping track of the players who are connected.

2.4.4 Remote-Procedure-Calls

Remote-Procedure-Calls, or RPC's for short, are functions that can be called over the network. These are the heart of sending events to other players. In order to mark a function as an RPC, the line ' [RPC] ' needs to be added onto the line above it, and this will have that function be listed in the Photon Server Settings file where the ID was added previously. The RPC's are sent via the PhotonView, so one will want to create a PhotonView variable into one's script that holds a reference to the object's PhotonView. In Project Runner's case the variable was named 'ScenePhotonView' simply to distinguish it from other PhotonViews that do not send RPC's. Making remote procedure calls is similar to sending regular message calls in Unity, in that the ScenePhotonView.RPC() receives the name of the function that is to be run as a string parameter. It also receives a parameter that defines the targets for the RPC (send to everyone, send to just the master client et cetera). The full call would be ScenePhotonView.RPC ('startRun', PhotonTargets.All); which would tell all connected players to run the startRun() function (Figure 6).

```
[RPC]
private void checkPlayerReady() {
    playerCheck += 1;
    if (playerCheck > PhotonNetwork.playerList.Length-1) {
        ScenePhotonView.RPC("startRun", PhotonTargets.All);
    }
}
```

Figure 6: Example of an RPC function and sending an RPC to all connected players

2.4.5 Coding a Data Synchronizer

While things like the Photon Transform View are useful for synchronizing objects across the network as far as their movement goes, some information will have to be synched manually. In Project Runner, this was the animation states. While Photon adequately moved all player characters in accordance to each character's owner's inputs, variables that told the `PlayerScript.cs` which animation to be playing (jumping, dashing, stumbling, running) for the other characters was not something that was being transferred. In order to send this data over the network, a new component was needed for it. The new script, named `NetworkPlayer.cs`, had boolean variables for each animation state, and a function that set the animations based on these booleans if the `PhotonView` of the character didn't belong to the current player. This way it's only run for the characters that are being updated through the network, and not locally. Now, the game simply needed to be able to get this information from the network, as well as send its own for the other players. Luckily, Photon comes with a preset function for just this, named `'OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info)'`.

`OnPhotonSerializeView()` is run every time the data packets are sent, and the information being sent is stored in the `PhotonStream` object. To have the function save information to the object, it first needs an if-statement to check whether the stream is currently writing to the network instead of reading from it. `'stream.isWriting'` will return true if it is currently writing. Placed inside that if-statement is the code `'stream.SendNext(myVariable)'`. This will have that variable serialized and sent over the network. Most basic data-types can be sent such as integers, booleans and strings, but objects cannot be serialized. Then placed in the 'else' part of the statement is `'myVariable = (dataType)stream.ReceiveNext();'` and this will read from the stream in the order that the data was written into it, and set whatever is read as the value of `'myVariable'`. This will need a cast for the read data that tells what the data-type is supposed to be for what is being read (Figure 7).

```

public void OnPhotonSerializeView(PhotonStream stream, PhotonMessageInfo info){

    if(stream.isWriting){
        //This OUR Player. Sending data

        stream.SendNext(isRunning);
        stream.SendNext(isJumping);
        stream.SendNext(isJumpingAgain);
        stream.SendNext(isStumbling);
        stream.SendNext(isDashing);
        stream.SendNext(currentAnimationState);
        stream.SendNext(transform.localPosition.z);

    }
    else{
        //Other Player. Receive data

        isRunning = (bool)stream.ReceiveNext();
        isJumping = (bool)stream.ReceiveNext();
        isJumpingAgain = (bool)stream.ReceiveNext();
        isStumbling = (bool)stream.ReceiveNext();
        isDashing = (bool)stream.ReceiveNext();
        currentAnimationState = (string)stream.ReceiveNext();
        transform.localPosition = new Vector3(transform.localPosition.x, transfo

    }
}

```

Figure 7: The OnPhotonSerializeView function of the Project Runner player character

3 Developing the Game

This chapter will be going through the process of developing the game, including introductions to a few core concepts and terms relevant to the development, and an in depth explanation of how the various mechanics of the game were created and implemented. The descriptions will give a better understanding of how to apply the tools available in Unity, and will explain more about how the created scripts interact with each other. Detailed descriptions of individual scripts will be found in chapter 5.

3.1 Unity Built-in Functions and Other Terminology

C# and various other programming languages have function called a 'foreach' loop, which is a variation of the 'for' loop. A foreach loop gets an array or a list as a parameter. The loop will go through each index in the array, referencing the current index by a name that is given as a parameter. 'foreach (GameObject g in objects)' will go through each index of the 'objects' array, and reference them as a GameObject type variable named 'g'. The programmers can then write the code for what they want to be done with every object in the array inside of the curved brackets that follow the foreach loop declaration. For example, when wanting to destroy every object in the 'objects' array, writing the line 'Object.Destroy(g)' inside of the

curved brackets will accomplish this. The foreach will go through each index and at each index run the code, which will in this case have Unity destroy the game object. (Unity Documentation.)

Every newly created scene in Unity comes with a default game object that is vital for the game, called the camera. When the program is run, the stuff shown by the program are the things that are visible to the camera object. The camera can have its size modified, be moved around and rotated for different viewpoints and be switched between 'orthographic' and 'perspective' modes, altering the way 'depth' is drawn (or rather, whether it is drawn at all or not). In perspective mode, everything is viewed from a single point, allowing the player to see the sides of objects that are for instance to the left of the center of the camera. In orthographic mode, the view is 'flat', and everything within the camera's scope is viewed straight-on. This way, the player will not see the side of an object even if it's to the left of them. This mode will effectively create a 2D view of the 3D game world. (Unity Documentation.)

One core component for essentially any type of game is the collider component. Simply adding a 3D shape to the game world will leave other 3D shapes capable of passing through it. While this is useful when intentionally wanting to combine multiple 3D shapes, it won't do for the player to be able to pass through the walls. This is where colliders come in. Colliders are 3D shapes that prevent other colliders from passing through them, unless an exception is specifically coded into the game. A collider itself is invisible, and is a separate component of a game object. It is possible to have a collider that isn't attached to a 3D shape, creating invisible walls. (Unity Documentation.)

There is also another type of collider called a 'trigger zone'. When a collider is made a trigger zone it will no longer stop objects from passing through it, though it will still stop a raycast. Instead, when detecting something contacting with the collider, it sends messages to its component scripts to call specific functions. The functions are 'OnTriggerEnter', which is called when something enters the trigger zone, 'OnTriggerExit', which is called when something exits the trigger zone and 'OnTriggerStay', which is called every time something that has entered the trigger zone continues to stay within it. Placing these functions into the collider's component scripts will have them be run during these events. (Unity Documentation.)

Unity also includes the Rigidbody component. Rigidbodies are Unity components that can be attached to game objects to allow them to be affected by Unity's physics engine. If one simply creates a cube into the game world, it will stay stationary unless ordered otherwise. However, adding a Rigidbody component to it will cause it to begin falling downwards as gravity takes effect until it comes into contact with a collider that can stop it. The Rigidbody will

also add in things like rotation upon impact with an angled surface. The component also gives access to a variety of built-in functions that can be used to apply force to the object from different angles. (Unity Documentation.)

It is also possible to define constraints for the Rigidbody, such as preventing it from ever rotating, or moving along the Z axis unless ordered otherwise through written code. The user can even disable all of the Rigidbody's ability to move by itself, reducing it back to a static object, but still giving access to more realistic acceleration and deceleration when moving it, by applying a certain amount of force and defining the object's 'mass'. In general, when wanting any game objects in the game world that can move, it is recommended to add a Rigidbody component to them. (Unity Documentation.)

Lastly there is the raycast function. A raycast is a line from one point towards a given direction. The raycast will be blocked by anything with a collider attached and that isn't specifically set to be ignored by raycasts. When the raycast is blocked, it will return a wealth of information, such as the distance between the hit and the starting point, the angle of the contact surface in relation to the angle of the raycast, the data of the object it hit and the location of the hit. (Unity Documentation.)

3.2 The Graphics and Audio

Marcus Dake was the graphics designer for the game for approximately a month into the game's development, during which time he created the animation sprites for one runner character, as well as multiple obstacles and terrain objects. After the first month however, he dropped out of the project to focus on school work, since his internship was about to be over anyway. After this point the graphics were also the author's responsibility, but most of the obstacles created after Dake's quitting of the project were done by utilizing the various artwork he'd made. They were merely adapted from his sprites into different types of terrain objects through combination and modification, in order to retain his original style.

While no proper audio (jumping sounds et cetera) was ever created for the game due to the lack of a sound engineer on the team, one piece of game music was created with the use of the Magix Music Maker program.

3.3 Development Phases

The first phase was creating the most basic aspects of the game so that it can be play-tested. A simple flat world was created by using a long stretched cuboid with a collision box to work as the floor, and placing a simple sphere with a Rigidbody attached to function as the 'play-

er'. The player object's rotation is frozen on all axes and its position on the Z-axis is locked. This way it can still move up, down, left and right, but it will never rotate or move towards or away from the camera unless a script is specifically made to make it do so. The next step was adding a few simple terrain objects that will be possible for the player to add, such as a 'mountain' which is a 45-degree tilted cube set half way 'underground' to create a protruding triangle, and a squished, sharp version to functions as a 'spike'.

In the player object's script the Update() function will always move the camera to a specific position in relation to the player, so that it always moved alongside the character, and attached a script for touch inputs to the camera. Additionally the player will be moved to the right at each update cycle by a set amount, and jump whenever a screen-tap was detected by the touchInput.cs script. Now there was a game world that the player could move around in and start testing the more advanced mechanics.

3.3.1 Detecting the Drawn Shape

The first challenge was to figure out how to have the game detect what kind of a shape the player 'draws' on the screen with their finger. At first, the seemingly easiest method was to create a grid of circular 'symbol' buttons that would highlight when the player drags their finger over them. Each of the buttons is told its coordinates on the grid, with 0,0 being the bottom-left symbol. Each time a symbol was highlighted, it would send its coordinates to the game controller, where they were stored into a list. Once the player lifted their finger away from the screen, the game controller would go through each index of the list, and check the changes in the coordinates from one symbol to another. If the X coordinate increased by one from the previous index, it would add 'ri' to a string, for 'right'. If it decreased it would add 'le' for left. The same was done for the Y axis changes, with 'up' for up and 'do' down.

With this, if the player for instance drew an upside down V with their finger, it would write 'riupriupridorido', with each 'riup' signifying a line towards the top right, and 'rido' signifying a line towards the bottom right. Once the game controller finishes going through the list, the list is emptied, and the resultant string is sent to a switch-case if-else combination that checks for matches, close matches and 'synonyms' (drawing the same shape but in a different order, such as backwards). If the string is equal to 'riupriupridorido', the Mountain terrain object is set as the 'selected' terrain object, and added a certain distance in front of the player.

3.3.2 Adding the Terrain

While simply adding a game object into the game world to a specific distance away from the player is quite simple, the challenge was figuring out the height at which to place it. While currently the game world's floor was a simple flatland, certain terrain objects would modify this, and eventually it was intended that it would have either a pre-generated or randomized starting terrain. So there needed to be some way to figure out what the 'ground-level' was at a specific point. The original attempt at accomplishing this was by creating an invisible falling game object that upon contact with the ground would add the selected terrain object at its contact point. However this caused a severe delay in the detection, since the object would have to fall from high enough to never spawn below any heightened terrain, while still falling slow enough that it doesn't simply 'phase' through the ground. This caused a one to two second delay, which while serviceable for testing, was quite annoying when trying to actually play. The issue was made worse by the inconsistency of the delay, making the game feel unresponsive.

The method was abandoned and replaced with a new way of doing it that was discovered, which used 'raycasts'. Raycasts are essentially lasers fired from a point in a specified direction, and will return a variety of information upon hitting a collider. This information includes the exact contact point coordinates in the game world. Not only was this method far simpler, but also instantaneous, allowing for no delay between finishing a shape and the creation of the corresponding terrain.

3.3.3 Dashing Through Objects

Initially it was attempted to have the player be able to use the 'dash' action to phase through obstacles by disabling the player's collision detection for the duration of the dash. The issue with this was that the player can then dash 'into' mountains and get stuck inside. So instead the dash was made to use the 3D element of the game to achieve the illusion of the player phasing through some game objects. The total width of the player in units is one, while the width of the game area is five. In order to allow the player to dash through some obstacles but not all, the dash was made to move the player 4 units farther away from the camera for the duration of the dash.

Because of the camera mode being set to 'orthographic', there is no visual change in the player being further away as distance is not being rendered. This however does allow the player character to pass behind certain obstacles that are made narrower, but not behind other wider obstacles. So for instance a mountain will be five units wide. This way, even after moving the player away, they still collide with the mountain. A spike, however, is only 2 units wide, meaning that the player will go behind the spike, but look like they're passing through it.

3.3.4 Looping the World

Up until this point the testing had been accomplished by doing runs from one end of the flatland to the other, and then simply rerunning the game. The intent however was to have the world 'loop' for at least a few laps, so that player created terrain will eventually come up in front of them, and each lap would make the game world more and more chaotic as more and more terrain and traps are spawned on top of each other. In order to accomplish this feature, 'boundary' walls were added at both ends of the flatland. Once a game object comes into contact with the boundary, it is teleported to the boundary at the beginning. While this achieves an infinitely looping game world, it is not nearly enough, as the teleportation is blatantly obvious. The challenge is to have the player appear at the beginning of the stage, without anything discernable happening from the player's perspective, creating the illusion that you are constantly running to the right without interruption.

The issue is that if a player is looking at another player reaching the 'end', the other player's character will disappear because they were teleported to the start, and only reappear once they themselves have also reached the boundary. In order to prevent this from happening, a 'clone' version of the player game object was added. This object is created at the boundary upon the player's teleportation, and will mimic all of the player's actions and animations. So long as the clone lines up perfectly with the other player, there will be no noticeable difference upon the player reaching the boundary and being teleported to the beginning. The same will have to be done with the boundary at the start, so that other players behind the player don't seem to disappear for a while once they hit the end boundary.

Once that was done, the same duplication effect needed to be added to the created terrain. Upon a terrain's creation, it checks if it is within a certain distance from one of the boundaries. If it is close enough, the terrain takes that distance, and has a duplicate of it created that same distance away from the other boundary. It took some time to synchronize everything correctly so that there is no 'jitter' happening at the moment of teleportation due to the duplicates not lining up perfectly. Once it was finished, however, the world now seemed to smoothly loop around itself without any indication of a 'teleport' happening.

3.3.5 More Advanced Draw Detection and Controls

The symbol grid was still a simple 5 by 5 locked in place in the middle of the game screen, with a jump happening anytime the screen was tapped. The placement of the terrain was still quite absolute, with ground modifications appearing the set distance away, traps a little further away and the height of a floating platform being dependent on the row on the grid that

the player drew the horizontal line on. After proper graphics were made for all of the traps used in the game, it was now time to make it all work more smoothly and intuitively. The grid was increased in size to a 10 by 10, with smaller more tightly placed circles. Instead of having the map locked into the center, and always visible but transparent, the symbol grid was made invisible and movable, appearing only when the screen is touched.

Code was added to the `touchInput.cs` that would detect the location of the very first touch event, take its X coordinate and check if it was far away enough from the left edge. If it was (approximately a tenth of the screen width), the symbol grid would be moved to be centered on that point. The grid would remain visible and active so long as the player kept their finger within the constraints. This way the player can draw their shapes anywhere on the screen, except behind their character, where there would be buttons for the jump and dash actions. Additionally, the location of the first symbol the player touched would now also be sent to the game controller, as this information is now what defines where terrain is added. The position of a floating platform is based on the first symbol's position in the game world at the moment the touch event ended, so it is possible for the player to draw the shape and then hold their finger on the screen for a while before releasing, and the platform will appear exactly where the shape they draw was at the moment they released. Ground objects will ignore the height, and simply look at the x coordinate, which is where the raycast will be shot from.

It is now possible to draw terrain quickly at essentially any point in front of the player character that's visible on the screen, and have the terrain's placement feel intuitive. There are also now dedicated buttons for the dashing and jumping on the left side of the screen, and a 'cooldown bar' at the top to visually indicate the time during which special actions cannot be performed.

3.3.6 Multiplayer

The basic game mechanics were now largely finished, and it came time to add in the multiplayer. Because there hadn't been any mention of Photon before this point, a large portion of the code did require rewriting in order to work properly in conjunction with the newly added Photon Networking Package. Most of it was quite simple however. Player and terrain objects had to be converted to utilize components such as the Photon Transform View in order to have player positions updated through the network to other players.

Most of these issues were handled as they came up during the addition of the matchmaking and network serialization code. The `Matchmaker.cs` script was added and it was responsible for connecting to the server, creating a room if one hadn't been created already, and giving

players their player numbers and starting positions. When players connect to a game, they will see the characters underneath the game title in the middle of the screen, with each character representing a connected player in the current game. Text at the top left corner would indicate how many players had pressed the 'ready' button. Once all joined players had pressed it, the Matchmaker.cs would tell the game controller to start the game. The Matchmaker.cs is also the script responsible for handling 'victory' conditions. The game controller would increase the player score for every unit of distance the player progresses, with extra points for laps. The matchmaker checks if the player has reached the score of 1500, which was the 'finish line'. When this is detected, the matchmaker sends a message through the network to all other players to inform them that a player has won, indicating the player by their player number (for example 'Player 1 has won!').

3.3.7 Alternate Version and Ending the Project

After the multiplayer and victory conditions were implemented, proper game testing could begin, with upwards of four people playing against each other, sometimes on different platforms. During the testing, doubts were raised as to how much fun the whole 'symbol drawing' aspect of the game actually was, particularly when most players had to be specifically told how to create each terrain formation. The idea was put forth to redesign the game without the symbol drawing function. Instead, add buttons for creating a simple hill (which would be the most commonly used terrain anyway), a spike, a platform and the special traps. For ease of comparison, the code was made so that the Game Controller had one tick box for whether or not the symbol grid was enabled. If not, the game would ignore the symbol grid, and instead add the new buttons that would call the terrain creation function directly. A pre-generated world was also added so that the first lap isn't just through flatland (Figure 8).



Figure 8: Screenshot of the game's alternate version with buttons for spawning obstacles

In the end, even the alternate version didn't really feel that enjoyable to play. The mechanic of adding terrain into the world to modify the track was an interesting one that hadn't been done before as far as we knew, but it simply didn't translate into a fun game experience. And once that was removed, all that was left is a standard speed runner type game with nothing else unique about it aside from maybe its art style. Additionally, the head of the project had come up with a more interesting idea for a game that showed more promise. Thus, Project Runner was scrapped in alpha.

3.4 Game Overview

When the scene is loaded, the MenuArranger, GameController and Matchmaker components of the Game Controller game object position everything in the screen and create the game object hierarchy. TouchInput begins to check for touch events. When detecting one, it will fire a raycast from the touch location and order any button type object hit by the raycast to run either the 'OnTouchDown', 'OnTouchStay' or 'OnTouchEnd' functions. Pressing the connect button has the ActionScript tell the Matchmaker to connect to the network and create the room. The MenuArranger switches the 'Connect' button to the 'Ready' button. Pressing it will send an RPC over the network to all connected players informing them that another player is ready. A player character is spawned on the screen to show how many players are ready. Once the number of ready players is equal to the number of connected players, Matchmaker tells the GameController to run the game start function.

The game start function will activate the player characters' PlayerScripts and remove the title and menu buttons. The PlayerScript will have the player character running at a set pace, increasing their score with every unit of distance travelled. When the 'jump' or 'dash' buttons are pressed, the PlayerScript is ordered to run the corresponding functions that will have the player change animation states and move accordingly. Touching anywhere on the middle of the screen will tell GameController to activate the 'symbol map'. Whenever dragging over any of the circular buttons on the symbol map, that button is disabled and its coordinate values added to a list in GameController. Once the touch event ends, GameController is told to disable the entire symbolmap and go through the list of coordinates. The coordinate changes from one button to the next are converted into a string of text, the list is cleared, and a switch-case if-else combination checks for matches. If a match is found, the corresponding terrain object is created at the x-coordinate of the touch event and the y-coordinate of either the touch event or the height of the ground (Figure 9).

Project Runner Flowchart

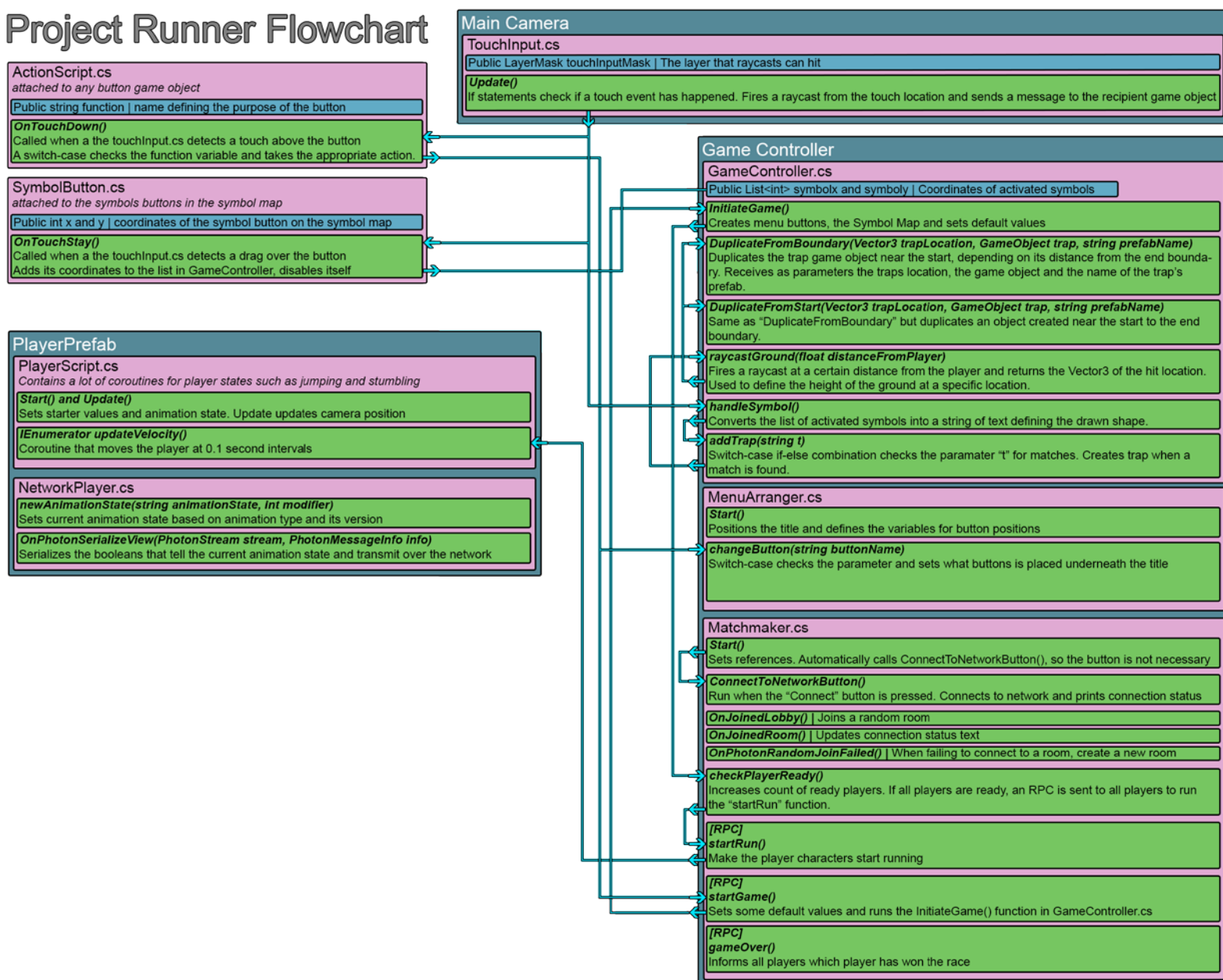


Figure 9: Project Runner Flowchart. Dark blue indicates a game object, purple a component, green a function and blue a variable.

4 The Game Code

This chapter will go through more detailed explanations of the game's mechanics, delving into and presenting the actual code itself, as well as some built-in functions of Unity that were very useful throughout the creation of the game.

4.1 Game Specific Mechanics and Terms

The game has a variety of mechanics with their own terms, so in order simplify the explanations for the scripts, this section will go through each of them.

4.1.1 Symbols, Traps and Terrain

'Symbols' are the shapes that you draw that define what type of game object is added to the game world, and the buttons on the central grid that are used to detect the drawn shape, are called 'Symbol Buttons'. Henceforth for the sake of brevity, the term 'symbol' will refer to the buttons, and the word 'shape' will be used to reference the drawing you make. The added game objects themselves are divided into two types. 'Terrain' and 'Trap'. A terrain object is any type of ground-like object that is added, that can be traversed by the player safely. This would be things like slopes, hills and floating platforms. Some objects on the other hand are not safely traversable and function purely as obstacles for the players. These objects are tagged as 'traps', and will cause the player to stumble upon coming into contact with them. Examples of these are things like walls and spikes.

4.1.2 Special Traps and Actions

In addition to being able create terrain, the player character can also perform various special actions, such as jump or dash through obstacles. Actions typically have their own separate corresponding button and a separate cooldown. One of these actions is a 'special trap' that the player can choose in the menu alongside their character. These traps possess a few unique qualities that the standard terrain and trap objects lack. One, they each have a separate cooldown to prevent them from being used too frequently. They also only exist in the world for a short amount of time (typically 5 seconds). Finally, they cannot be used by the player who is in the lead, as their purpose is to give players at the back a chance to catch up.

While they are called 'special traps', all of them are not obstacles. The 'Cage' encircles the lead player, and the 'Spikes' adds five large spikes. However, the 'Warp' adds a glowing vertical line in front of the player who used it for five seconds. Any player who passes through the warp gains a massive speed boost for two seconds, during which they can freely phase through obstacles.

4.2 Setting Up the Project

While the game will be played as a 2D side-scroller, the actual game environment will be a 3D game world. As such, the Unity project is set up as a 3D project. The reason for this is that while essentially everything that can be done in a 2D Unity project can be done in a 3D project, a 2D project lacks a lot of the tools available for a 3D project. In order to make the 3D

game world appear two dimensional, the camera game object is set to 'orthographic projection' instead of 'perspective'.

I then added the Game Controller game object that would hold the GameController.cs script. The game controller handles all of the general and game-world related code, such as the default values, game options, menu placements, network connections and the music. The Game Controller object's position is meaningless as it's merely a container for all of the scripts and other components, but many tend to place it at 0,0 in the game world for simplicity.

The project directory has folders created for things that would be used throughout the development. 'Animations' for all of the sprite animations. 'Textures' for all of the in-game graphics. 'Audio' for the music and other possible sound effects. 'Materials' for the 3D shapes that were in particular used during early phases when textures weren't available yet and all game objects were made from Unity standard 3D shapes. 'Photon Unity Networking' and 'Plugins' would hold all of the Photon package related resources. 'Scripts' for all of the code, and 'Prefabs' for all of the premade game objects that would have clones of them created into the game world, such as terrain objects. '_Scenes' is the folder that contains the separate scenes of the game. The '_' is at the beginning simply so the alphabetical order of the folders places the scene folder at the top.

4.2.1 Scenes

The game is divided into two separate scenes. 'Main' is the scene in which the actual game-play occurs. It contains all of the code for the online connection and matchmaking, and a game controller designed for handling the events of the game. 'Menu' is the other scene, which is run first, and has the player choose their game options. It also includes an 'instructions' portion that shows what shape corresponds with what terrain object. Once the player chooses their character and special trap, the program loads the 'Main' scene.

4.2.2 Prefabs

All of the buttons used in the game are saved as prefabs, as are all of the terrain objects and the player. The terrain objects are created by adding the proper graphic to a SpriteRenderer component, and then placing multiple 3D colliders in such a manner that they mostly follow the intended 'surface' of the terrain object. The width of the collider depends on the type of terrain and whether or not it can be 'dashed' through. Spikes are narrower, allowing the player to essentially dash 'behind' them, while ground objects are wider to prevent the player from being able to pass through them.

4.3 Scripts

This is a short summary of all of the scripts in the game, while the other sub-chapters will contain more in-depth descriptions of the major scripts. The `ActionScript.cs` is the script responsible for handling buttons presses and their functions. `MenuArranger.cs` takes care of arranging the buttons and graphics in the Menu scene, while `MenuController.cs` handles the functionality. `MoveAtBoundary.cs` is the script attached to the boundary at the end of the game area which teleports the player to the start. `ScoreHandler` contains the coroutine for increasing the player score based on distance travelled. `StaticVariables.cs` holds the game options as static variables, stored outside any particular scene. This is used to have options selected in the Menu scene carry over to the Main scene. `SymbolButton.cs` is the script that handles the buttons on the symbol grid. `TouchInput.cs` and `TouchInputMenu.cs` (Menu scene specific) process screen touches and, on the PC testing version, mouse events. `TrapButton.cs` adds terrain graphics to buttons that are supposed to add terrain to the world when pressed, to work as visual indicators of the button's function. `GameController.cs` is the largest script, attached to the Game Controller game object. It creates the world and handles the majority of the game functions, including the addition of terrain. `Matchmaker.cs` is the script that connects to the network and handles the majority of the network communication. `Network-Player.cs` is responsible for updating the player character position across the network. `PlayerScript.cs` contains all the code for moving the player character and animating it. `Player-Clone.cs` is the script attached to the copy of the player created at the boundary that mimics all of the player character's movements.

4.3.1 GameController.cs and MenuArranger.cs

The `GameController.cs` is one of the central scripts for the game. It contains 500 rows of code, and possesses references to all of the prefabs used in the game, variables for the game options such as the size of map, starting locations et cetera and some default values that are used during the game's running time, such as the quaternion 'rotation', which is a simple null rotation value for objects that require a rotation value to be given, but don't need to be rotated.

The `'Start()'` function only contains an order to prevent the mobile device from ever darkening the screen while the game is playing, as the actual setup of the game is done in `Matchmaker.cs` and in the Unity game world editor screen. The actual starting of the game is done by the `'InitiateGame()'` function, called from `Matchmaker.cs` when all connected players are ready. `InitiateGame()` sets the buttons in their proper locations and creates the symbol grid. The rest of the functions will either be covered in the later chapters that deal with other

scripts that utilize the functions in GameController.cs, or are just simple functions for handling various bits of data or the buttons.

The MenuArranger.cs is a simple script that positions the title and starting buttons when the scene is loaded, and has the 'changeButton()' function for changing the 'Connect' button to the 'Ready' button and then to the 'Start' button. The button changer is used by Matchmaker.cs to progress through the connection and setup stages. Some of the menu arranger's variables are also used in the 'InitiateGame()' function of the game controller.

4.3.2 Matchmaker.cs and NetworkPlayer.cs

Matchmaker.cs is the other core script of the game. It uses GameController.cs and MenuArranger.cs to handle all of the 'pregame' functions such connecting to a network and waiting for players to join and start the game, and all the buttons and graphic changes involved. It connects to the server and checks if a 'room' is already created. If not, it will create one and join it. The progress of the connection is detailed underneath the title (Figure 10). Whenever a player joins a room, a new player character is created, and the character is given a number to identify them based on the order they joined the room. When a player presses the 'ready' button, all other members of the room are informed of this. Once the number of ready players is equal to the number of joined players, the 'master', the first person to have joined, is informed of this. The master then send a remote procedure call to all the other players ordering them to run the 'startGame()' function. The matchmaker also handles all the players' scores and the 'game over' event.

```
private IEnumerator connecting(){
    while (PhotonNetwork.connectionStateDetailed != PeerState.Joined) {

        ma.connectionText.text = PhotonNetwork.connectionStateDetailed.ToString();
        yield return new WaitForSeconds(0.1f);
    }
    ma.connectionText.text = "";
    ma.changeButton ("ready");
    Debug.Log ("Number of characters added: " + charactersAdded);
    updateServerText ();
}
```

Figure 10: The coroutine that connects to the server and prints text about its progress at 0.1 second intervals.

The NetworkPlayer.cs is attached to the player character, and handles the transference of player data to other players. While PlayerScript.cs actually moves and animates the player, it informs NetworkPlayer.cs of its current location and animation state, and the OnPhotonSerializeView() function in NetworkPlayer.cs then communicates that to all of the other players so that their local versions of the other characters are animated correctly.

4.3.3 PlayerScript.cs and PlayerClone.cs

The PlayerScript.cs holds the code for the actual player character that moves in the game world. The updateVelocity() coroutine constantly pushes the player forwards at a set velocity, while the Update() function moves the camera alongside the player. The X position of the camera is always kept at a certain distance of the player, whereas the Y position is only changed if the player jumps high enough (Figure 11). It also has coroutines for checking whether the player is grounded or stuck. The 'grounded' variable is used in changing the animation states and counting how many times the player can jump, while the 'stuck' variable is used to cause the player to be teleported up and forwards if the player has not moved for 3 seconds. An OnTriggerEnter() function causes the player to stumble when contacting with a 'trap' type terrain object, slowing the player down and changing the animation for the duration of the slowdown. It also has functions for all of the player actions and for changing any of the animation states, with a separate function for each action such as jumping (Figure 12).

```
void Update () {
    if (photonView.isMine){
        //If player jumps high enough the camera locks on to his position

        if(rigidbody.velocity.x < -10){
            rigidbody.velocity = new Vector3 (10, 0, 0);
            //StartCoroutine(crash());
        }

        if (rigidbody.position.y > 4) {
            camYoffSet = -2 + rigidbody.position.y/2;
        }else if (rigidbody.position.y < -2){
            camYoffSet = 2 + rigidbody.position.y;
        }else{
            camYoffSet = 0;
        }

        camXoffSet = 10;

        Vector3 moveCam = new Vector3 (transform.localPosition.x + camXoffSet, camYoffSet + 4, Camera.main.transform.localPosition.z);
        Camera.main.transform.localPosition = moveCam;
    }
}
```

Figure 11: The Update() of the PlayerScript.cs

```

public void jump(){
    if(jumpCount < 2){
        np.newAnimationState("jumping", jumpCount);

        if(jumpCount == 1){animator.SetTrigger("jumpingAgain");}
        else{animator.SetTrigger("jumping");}

        animator.SetBool("isGrounded", true);

        jumpCount++;
        rigidbody.velocity = new Vector3(rigidbody.velocity.x, 10, 0);

        animator.SetBool("isGrounded", false);
    }
}

```

Figure 12: The function for jumping. Only allows two jumps before needing the player to touch the ground. Resets the “isGrounded” boolean and add upwards velocity.

PlayerClone.cs is attached to the created copy of the player character. It possesses the same variables as the PlayerScript.cs, but only has the Update() function for changing its own position and animation state to correspond with the original’s, and the ‘setupPlayer()’ function for defining which character is the original. Like the original, the clone also has the NetworkPlayer.cs attached for updating its position through the network.

4.3.4 TouchInput.cs for Detecting Screen Touches

The buttons are 3D game objects with their position tied to the camera, so that they move alongside the camera as the game scrolls. In order to have the game detect whenever the screen is pressed on the position of one of the buttons, the author firstly needed to add a new layer that the buttons belonged to, named ‘touchInput’. All buttons would be assigned to this layer. Additionally the buttons will have the script ‘ActionScript.cs’ attached to them, and the camera will have ‘touchInput.cs’ script.

The touchInput.cs script has the public ‘LayerMask’ type variable ‘touchInputMask’, which will be assigned to be the aforementioned ‘touchInput’ layer. This will be used later to make sure that only game objects in the ‘touchInput’ layer are detected by the script. Every time the game updates, the script needs to check if the screen has been touched. Fortunately Unity already has its own function for this, called ‘Input’, which handles keeping track of all input given to the software. It includes things such as an array with each screen touch event in it, as well as a counter for the number of touches. We can see if the screen has been touched simply by checking if ‘Input.touchCount’ is greater than zero. If it is, the script will then proceed to run a foreach loop for the ‘Input.touches’ array, sending a raycast from the position of each of the touches. The camera-view is the ‘Screenspace’, where 0,0 is bottom-left cor-

ner of the camera and the top-right corner is the width and height of the camera. 'camera.ScreenPointToRay' creates a raycast in the game world based on the camera position in the world and the 'screenspace' value given, firing the ray in the direction the camera is looking (Figure 13).

```

else if (Input.touchCount > 0) {
    foreach (Touch touch in Input.touches) {

        Ray ray = camera.ScreenPointToRay (touch.position);

        if (Physics.Raycast (ray, out hit, touchInputMask)) {
            GameObject recipient = hit.transform.gameObject;

            if (touch.phase == TouchPhase.Began) {
                recipient.SendMessage ("OnTouchDown", hit.point, SendMessageOptions.DontRequireReceiver);
            }
            else if (touch.phase == TouchPhase.Moved) {
                recipient.SendMessage ("OnTouchStay", hit.point, SendMessageOptions.DontRequireReceiver);
            }
            else if (touch.phase == TouchPhase.Ended) {
                gc.handleSymbol();
            }
        }
    }
}

```

Figure 13: Part of the touchInput.cs. Checks touches, shoots raycast, sends message

The raycast is given the touchInputMask as a parameter, so that it only returns a 'hit' when the ray hits a collider assigned to a game object with the assigned layer (in this case, the 'touchInput' layer). If a hit is registered, the game object that was hit is set as the 'recipient', and the script sends a message to the game object. The message contains the name of the function that needs to be run as its first parameter. The name is defined based on the type of the 'touch'.

When the script registers that a touch has just begun, it will send a message to the 'On-TouchDown' function of the recipient object. If it detects that after the touch began, the user began moving, instead of lifting their finger, it will call the function 'OnTouchStay'. This will be used to record the 'shape' that the user draws with their finger. Once the script detects that the touch event has ended, it will call the 'handleSymbol()' function in the game controller script, which is responsible for checking that the drawn shape corresponds with a predefined shape, and also adding new terrain based on that shape. Since not all of the buttons will necessarily include both the 'OnTouchStay' and 'OnTouchDown' functions, the message will also have 'SendMessageOptions.DontRequireReceiver' as a parameter. This way there is no error if someone holds their finger over a button that lacks the 'OnTouchStay' function.

4.3.5 ActionScript.cs for Handling Buttons

The ActionScript.cs is the script attached to the buttons that handles their functions. The script has the string variable 'function'. Once OnTouchDown() is called, a switch-case checks

what the 'function' variable is and does the appropriate actions. For example, if the function on a button is defined as 'Exit', the script will run 'Application.Quit()' to shut down the program, or if it's 'Jump', the jump() function in the PlayerScript.cs is run. The script also has multiple cooldown variables. The 'functionPause' (set to 0.5 seconds) prevents too many button inputs from being received simultaneously. 'actionCoolDown' prevents the player from doing too many special actions (such as dashes) too many times in a row, and the 'trap-CoolDown' prevents the player from creating too many special traps too fast.

The pauses are created by defining the 'nextAction' variable as the current time (Time.time) plus the defined action pause (ex. '0.5f' for 0.5 seconds). When OnTouchDown is called, it checks if the current time is greater than 'nextAction'. If 0.5 seconds have passed since the last action, it will be. For example, if the game has been running 10 seconds and the player does an action, the 'nextAction' variable will be set to 10.5 seconds. If the player were to attempt an action 0.2 seconds after the first, the game time would be 10.2 seconds, and thus the attempt would be ignored (Figure 14).

```
if(Time.time > nextAction){
    nextAction = Time.time + actionCoolDown;
    p.action ();
}
```

Figure 14: setting the cooldown

4.3.6 MoveAtBoundary.cs and Object Duplication in GameController.cs

At the end of the game area there is placed an invisible wall with a trigger collider and the 'MoveAtBoundary.cs' script. The script has the 'OnTriggerExit' function and a 'boundaryEnabled' boolean variable. 'OnTriggerExit' is called twice when the player travels through the trigger zone. On the first run it will set 'boundaryEnabled' to false, and check if the colliding object belongs to the player by looking at its Photon View's 'isMine()' function. If it is, the player score is increased for completing a lap. If not, it checks if the object is tagged as a 'player'. When it is, which means it's the character of another player, a copy of that player is created to hide it teleporting to the start of the game world, making it seem like they continue running unabated. The copy will be destroyed after 5 seconds, giving it enough time to travel beyond the visibility of other players. Once this is done, a clock is started that will re-enable the boundary in 0.2 seconds. Before this happens however, the OnTriggerExit is called again. Because the boundary is set to disabled, none of the earlier code is run, and instead it simply checks if the object is a player character. If it is, that character is teleported to their starting position (Figure 15).

```

void OnTriggerExit(Collider other){
    if(boundaryEnabled){
        Debug.Log (other);
        boundaryEnabled = false;
        if(other.GetComponent<PhotonView>().isMine){
            sh.Lap = sh.Score;

        }else if(other.tag == "Player"){
            StartCoroutine(copyPlayer(other.transform.root.gameObject, other.transform.localPosition));
            other.transform.localPosition = new Vector3(gc.playerStartPosX, other.transform.localPosition.y, gc.playerStartPos.z);
            other.GetComponent<ScoreHandler>().Lap = other.GetComponent<ScoreHandler>().Score;
        }
        StartCoroutine(enableBoundary());
    }
    else if(other.tag == "Player"){
        other.transform.localPosition = new Vector3(gc.playerStartPosX, other.transform.localPosition.y, gc.playerStartPos.z);
    }
}

private IEnumerator enableBoundary(){
    yield return new WaitForSeconds(0.2f);
    boundaryEnabled = true;
}

public IEnumerator copyPlayer(GameObject other, Vector3 otherPos){
    yield return new WaitForSeconds (0.1f);
    GameObject temp = Instantiate(playerClone, new Vector3(otherPos.x + 0.5f, otherPos.y, otherPos.z), other.transform.localRotation) as GameObject;
    temp.GetComponent<PlayerClone>().setupPlayer(other);
    temp.GetComponentInChildren<TextMesh>().text = "" + other.GetComponent<PhotonView>().ownerId;
    StartCoroutine(removeClone(temp));
}

```

Figure 15: OnTriggerExit and the player copying script

While the MoveAtBoundary.cs handles teleporting the player from the end of the game area to the start, this in of itself would still leave the teleportation noticeable. In order to further the illusion, both the start and end of the game area have to look identical. The MoveAtBoundary.cs takes care of creating a clone of the player that continues running after the original has been moved away, while the functions for handling added terrain is in the GameController.cs. The task of copying created terrain is divided into two functions. 'duplicateFromBoundary' and 'duplicateFromStart'. Whenever a new piece of terrain is added, the game controller checks whether or not it is within 20 units of either the boundary or the starting position. If either is true, the appropriate function is called.

Both of the functions receive the 'trapLocation' Vector3 variable, the terrain game object and the game object's name as parameters. The trap location is used to calculate the distance between it and the boundary. Once this distance is known, another instance of the object will be created that same distance away from other boundary. If an object is created 10 units in front of the end boundary, a copy of it will be placed 10 units in front of the starting position (Figure 16). Both functions also check if the object being copied is the 'warp' terrain object. When true, the functions will also start the 'removeTrap' coroutine, which will have the warp removed in 5 seconds, just like its original.

```

private void duplicateFromBoundary(Vector3 trapLocation, GameObject trap, string prefabName){
    float x = playerStartPosX - 1.1f + (trapLocation.x - boundaryLocation.x);
    float y = trapLocation.y;
    float z = trapLocation.z;
    GameObject temp = PhotonNetwork.Instantiate (prefabName, |
                                                new Vector3(x,y,z),
                                                trap.transform.localRotation, 0);
    temp.transform.localScale = trap.transform.localScale;
    if (prefabName == "Warp") {
        StartCoroutine(removeTrap(temp, 5));
    }
}

private void duplicateFromStart(Vector3 trapLocation, GameObject trap, string prefabName){

    float x = playerStartPosX + 19.9f + trapLocation.x + boundaryLocation.x;
    float y = trapLocation.y;
    float z = trapLocation.z;
    GameObject temp = PhotonNetwork.Instantiate (prefabName,
                                                new Vector3(x,y,z),
                                                trap.transform.localRotation, 0);
    temp.transform.localScale = trap.transform.localScale;
    if (prefabName == "Warp") {
        StartCoroutine(removeTrap(temp, 5));
    }
}

```

Figure 16: The duplication functions in GameController.cs

4.3.7 SymbolButton.cs and Trap Addition in GameController.cs

The SymbolButton.cs is attached to all of the symbol buttons in the grid. A double 'for' loop in the Start() of GameController.cs creates the grid of the symbols, defining the variables specific for each button (such as coordinates on the grid) and spacing the buttons out based on the defined variable (Figure 17). SymbolButton.cs has variables for its X and Y coordinates, whether or not it is enabled (It will not be after you've pressed it once, so you can't activate the same symbol button twice) and its graphics (it will be faded by default, and highlighted when selected). The Update() will check if the symbol button is enabled, in which case it will set its graphic to 'faded' to signify that it isn't selected anymore. It also has the 'On-TouchStay' function, which is called when the player holds their finger on the screen and 'draws' over the button. When called, it will check if the button is enabled. If it is, it will be disabled, its color highlighted and its coordinates sent to a list in GameController.cs.

```

//INSTANTIATES SYMBOL BUTTONS
for (int y = 0; y < symbolMapRangeY; y++) {
    for (int x = 0; x < symbolMapRangeX; x++) {
        pos = Camera.main.ScreenToWorldPoint(new Vector3((Screen.width/2 + Screen.width/4 - spacing*2) + (spacing*x),
                                                         Screen.height/2 - spacing + (spacing*y), 7));
        GameObject symbol = Instantiate(SymbolButton,pos,rotation) as GameObject;
        symbol.transform.localScale = new Vector3(1.2f, 1.2f, 1);
        symbol.GetComponent<SymbolButton>().x = x;
        symbol.GetComponent<SymbolButton>().y = y;
        symbol.transform.parent = symbolGraphicMap.transform;
        symbolMap[x,y] = symbol;
    }
}

```

Figure 17: Double 'for' loop for creating the symbol grid

When a touch event ends, the 'handleSymbol()' function in GameController will be called. The function goes through the lists of selected symbol button coordinates (symbolx and symboly for the x and y coordinates respectively), use the changes in coordinates from one button to another to write a string (Figure 18). This string is used to check if the drawn shape corresponds with a predefined shape in the 'addTrap' function, which receives the string as a parameter. After this the list is cleared in the 'resetSymbol()' function, that also re-enables all the symbol buttons and hides the symbol grid.

```
public void handleSymbol(){
    if(symbolx.Count > 0){
        trap = "";

        int y = symboly[0];
        int x = symbolx[0];

        for(int i = 1; i < symbolx.Count; i++){
            if(symbolx[i] == x+1){ trap += "ri";}
            else if(symbolx[i] == x-1){ trap += "le";}
            if(symboly[i] == y+1){ trap += "up";}
            else if(symboly[i] == y-1){ trap += "do";}
            y = symboly[i];
            x = symbolx[i];
        }
        Debug.Log ("You wrote: " + trap);

        if(trap != ""){
            addTrap(trap);
        }
    }
    resetSymbol();
}
```

Figure 18: handleSymbol() in GameController.cs

The AddTrap() function contains a switch-case that compares the received string with predefined cases for exact matches. If no exact match is found, it progresses to an if-else that checks for near matches and alternative ways of writing out the same shape. If a match is found the corresponding game object will be added to the game world (Figure 19).


```

public void addTrap(string t){

    GameObject temp = null;
    Vector3 pos = new Vector3(0,0,0);
    Vector3 ray = new Vector3(0,0,0);
    string trapName = "";
    float trapSpawn = 0;
    float groundSpawn = 6;
    float warpSpawn = 19;

    switch (t){
    case "riupriupridorido":
        Debug.Log ("LARGE MOUNTAIN SYMBOL");
        ray = raycastGround(groundSpawn);
        trapName = "Mountain";
        pos = new Vector3 (ray.x, ray.y+2, 0);
        temp = PhotonNetwork.Instantiate(trapName, pos, rotation, 0);
        break;
    }
}

```

Figure 19: addTrap() function part in GameController.cs.

5 Conclusions and Author's Comments on the Project

When the game finally reached a phase where it could be properly play-tested, there were no problems with the actual programming, and there was nothing but praise for how fast the game had been developed to this point. However, the gameplay itself simply wasn't found to be that interesting. It was a possible outcome that had been considered from the very beginning. When the concept for the game was created, one of the reasons we wanted to really try it out was that there were no other games like it. None of us knew of a multiplayer runner game where you modified the terrain on the go by drawing on the screen. Maybe we came up with something new and innovative. Or maybe, there was a reason why no one had done it before.

As it turned out, while an interesting idea, it simply did not translate to very fun gameplay. While it works, it's not something that provides a gripping gaming experience. It is not something that would catch the attention of people and be financially successful. When compared to other four person multiplayer runner games such as 'Speed Runners', the game falls sorely short by its very concept. While a few varying versions were made, in the end it was concluded that the entire project will simply be scrapped. No one believed it was really worth the trouble of trying to improve it.

While I do still have the game's code, and I was given essentially full ownership of it to do with it whatever I want, I doubt I will ever continue development of the game. As such, the game ended in what could at best be referred to as an 'early-alpha' build. Despite this however, the project was an excellent introduction to the usage of Unity, Photon and C#, and I garnered a lot of valuable experience in programming and game development during the

course of it. While it is unfortunate that the game idea itself didn't end up being particularly good, the lessons learned from it were well worth it. While tutorials and such are all well and good for getting started, the best way to really learn how to apply a tool is to work on a real project that involves it. The game forced me to create solutions for problems that I hadn't had to deal with in my prior experiences with programming, such as creating a looping world, animating sprites and complex touch detection.

There are various things related to code optimization and in general to the creation of cleaner code that I know now and would do differently were I to do it all over again. And my overall feelings about Unity are that it is an excellent tool for creating interactive software. It was quite different from Java and Eclipse that I had used before, so it took some time to really internalize how it worked, but I will definitely be using the program in the future if given the option.

References

2015. Unity Technologies. Company Facts. Retrieved 19.11.2015.
<http://unity3d.com/public-relations>

Allain, A. 2011. What's the point of C#? Retrieved 20.11.2015.
<http://www.cprogramming.com/tutorial/csharp.html>

Chapman, S. 2014. What is JavaScript? Retrieved 24.11.2015.
<http://javascript.about.com/od/reference/p/javascript.htm>

Jones, B. 2001. What is C#? Retrieved 20.11.2015.
<http://www.developer.com/net/asp/article.php/922211/What-is-C.htm>

Media.wiley.com. 2014. What is JavaScript? Retrieved 24.11.2015.
http://media.wiley.com/product_data/excerpt/88/07645790/0764579088.pdf

Mozilla. 2015. JavaScript Introduction. Retrieved 24.11.2015.
<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction>

Photon. 2015. Photon PUN. Retrieved 24.11.2015. <https://www.photonengine.com/en/PUN>

Photon. 2015. Photon Unity Networking Intro. Retrieved 24.11.2015.
<https://doc.photonengine.com/en/pun/current/getting-started/pun-intro>

tonyd. 2009. Newbie guide to Unity JavaScript (long). Retrieved 24.11.2015.
<http://forum.unity3d.com/threads/newbie-guide-to-unity-javascript-long.34015/>

Unity Documentation. Retrieved 19.11.2015. <http://docs.unity3d.com/Manual/index.html>

Figure

Figure 1: Two 'JavaQuest II' art style examples	10
Figure 2: Concept art of the standard 'Runner' character by Marcus Dake	11
Figure 3: The game object hierarchy.....	16
Figure 4: The Inspector showing the components attached to the Game Controller	17
Figure 5: The folder hierarchy with saved prefabs	18
Figure 6: Example of an RPC function and sending an RPC to all connected players.....	25
Figure 7: The OnPhotonSerializeView function of the Project Runner player character ...	27
Figure 8: Screenshot of the game's alternate version	35
Figure 9: Project Runner Flowchart	37
Figure 10: The coroutine that connects to the server.....	41
Figure 11: The Update() of the PlayerScript.cs	42
Figure 12: The function for jumping	43
Figure 13: Part of the touchInput.cs. Checks touches, shoots raycast, sends message	44
Figure 14: setting the cooldown	45
Figure 15: OnTriggerExit and the player copying script	46
Figure 16: The duplication functions in GameController.cs.....	47
Figure 17: Double 'for' loop for creating the symbol grid.....	47
Figure 18: handleSymbol() in GameController.cs	48
Figure 19: addTrap() function part in GameController.cs.....	49